An Integrated Approach to Engineer and Enforce Context Constraints in RBAC Environments

MARK STREMBECK and GUSTAF NEUMANN
Vienna University of Economics and BA

We present an approach that uses special purpose role-based access control (RBAC) constraints to base certain access control decisions on context information. In our approach a *context constraint* is defined as a dynamic RBAC constraint that checks the actual values of one or more contextual attributes for predefined conditions. If these conditions are satisfied, the corresponding access request can be permitted. Accordingly, a *conditional permission* is an RBAC permission that is constrained by one or more context constraints. We present an engineering process for context constraints that is based on goal-oriented requirements engineering techniques, and describe how we extended the design and implementation of an existing RBAC service to enable the enforcement of context constraints. With our approach we aim to preserve the advantages of RBAC and offer an additional means for the definition and enforcement of fine-grained context-dependent access control policies.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Elicitation Methods, Methodologies*; D.2.9 [**Software Engineering**]: Management—*Life Cycle, Software Process Models*; D.4.6 [**Operating Systems**]: Security and Protection—*Access Controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized Access*

General Terms: Design, Management, Security

Additional Key Words and Phrases: Context-dependent access control, context constraints, constraints engineering, role-based access control

1. INTRODUCTION

The evolution of software and hardware technologies for interactive networked applications is progressing at a high pace. This poses high demands on access control services that are deployed in interconnected and interactive

A version of this paper appeared in the *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003).*
Authors' address: M. Strembeck and G. Neumann, Vienna University of Economics and BA, Department of Information Systems, New Media Lab, Augasse 2-6, 1090 Vienna, Austria; email: mark.strembeck@wu-wien.ac.at, gustaf.neumann@wu-wien.ac.at

environments. In particular, such services often need to consider context information to enforce complex access control policies that rely on information like time, location, process-state, or access history, for instance. Therefore, permissions and permission assignment often depend on such context information. One possibility to deal with a dynamically changing context is to rapidly modify permission assignment relations according to the changes in the environment. An other approach is to define conditional permissions, that is, permissions that consider certain context conditions in access control decisions and thus are context-aware to a certain degree. Either way, it is sensible to adapt existing access control models and technologies to meet the needs of networked interactive applications, as offered by web-based services and pervasive computing environments for example. Therefore, we think that an access control mechanism with context constraints should be based on well-known models and techniques, and should offer a path from "traditional" to context-dependent access control policies.

Role-based access control (RBAC) [Ferraiolo et al. 2001; Sandhu et al. 1996] provides an access control model that enables the enforcement of many different access control policies. A central idea is to support constraints on almost all parts of an RBAC model (e.g., permissions, roles, or assignment relations) to achieve a high flexibility. Static and dynamic separation of duties are two of the most common types of RBAC constraints [see, e.g., Ahn and Sandhu 2000]. However, as mentioned above, it is often required to consider various context information in authorization decisions, especially in highly interconnected and interactive environments. Moreover, in many real-world applications it is necessary to enforce fine-grained policies where permissions are directly assigned to certain individuals [see, e.g., Adam et al. 2002; Georgiadis 2001].

In this paper, we propose context constraints as a means to consider context information in access control decisions. We present a process for the engineering of context constraints, which is designed as an extension to the scenario-driven role engineering process [see Neumann and Strembeck 2002]. Moreover, we describe how we extended the design and implementation of the xoRBAC component [cf. Neumann and Strembeck 2001) to enable the enforcement of context constraints. With this extension, xoRBAC provides an access control service that preserves the advantages of RBAC [see e.g., Sandhu et al. 1996] and allows for the definition of "traditional" RBAC policies. Additionally, it adds further flexibility through the specification of fine-grained context-dependent access control policies via context constraints.

1.1 Motivation

In recent years, software-based appliances and applications rapidly evolved from relatively isolated stand-alone computer workstations to interconnected and highly flexible devices that are used in almost any part of human life. Together with the widespread deployment of the respective technologies corresponding security requirements arose to protect sensitive services and information objects that are (potentially) accessed by many interacting users and/or machines. One important demand in this respect is the enforcement of customized context-dependent access control policies.

Some motivating examples of applications that inevitably need to consider context information in authorization decisions are sketched below:

—In the area of computer-supported cooperative work (CSCW) such as workflow management, or groupware applications, context, may, for example, consist of the interacting persons, the processed documents, the daytime, the logical and/or physical location of a person, and so on. Researchers have already investigated related access control (and other security) issues for more than two decades and achieved many improvements [see, e.g., Bertino et al. 1999, 2001; Georgiadis et al. 2001]. However, even this relatively well-known area still offers a rich field for further research.

—Mobile code applications range from comparatively simple downloaded Java applets to proactive mobile agents that gather information from distributed sources and/or autonomously travel in a computer network and react on certain events. The protection of host computers from malicious mobile applications, as well as the protection of mobile applications from malicious hosts, results in many access control related problems where context information needs to be considered, for example, the owner of an agent, the owner of a host, the access/travel history of an agent and so on [see, e.g., Edjlali et al. 1998; Jaeger et al. 1999].

—Another example is the purchase of digital goods over the internet, for example, downloading research articles from digital libraries, purchasing music files directly from an artist, or subscribing to a video streaming channel. Digital goods may pass into the possession of the respective customer, where the owner may use the corresponding products as often, or as long, as she likes to. However, one may also sell only a restricted number of uses or limit the authorized users to some explicitly named individuals. This and other context information may be captured in special digital contracts for instance [see, e.g., Guth et al. 2003]. Although some recent contributions describe sophisticated approaches for specific subdomains, [e.g., Adam et al. 2002] the whole field is still young and a large number of open research questions remains.

—Hardware technologies for wireless communications as Bluetooth, Wireless LAN (IEEE 802.11), or mobile phone related technologies distribute quickly. Moreover, middleware standards such as CORBA or the simple object access protocol (SOAP), and software technologies for dynamic service lookup and ad hoc networking like Jini, universal plug and play (UPNP), or E-Speak evolve [see, e.g., Kim et al. 2002]. With these technologies the vision of ubiquitous and pervasive computing [Weiser 1991; Weiser 1993] is about to become reality. They enable the realization of novel applications based on mobile devices [see, e.g., Schmidt et al. 1999]. For example, customized location-based services, distance monitoring of medical parameters, direct and ad hoc interactions through mobile devices, or an intelligent/aware home that responds to particular events and actively controls the access to certain services [see, e.g., Covington et al. 2001].

The impressive technical opportunities yet give also an enormous rise to complexity of information security in general and of access control in particular. For

example, publicly offered services (commercial as well as nonprofit) must be protected so that only authorized users may access specific resources. Furthermore, user-related information needs to be protected from illegal accesses, no matter if the respective information is stored on a user's mobile device, through an intelligent home environment, or by a publicly available service (such as connection or movement logs of cell phones). Among other things, the fulfillment of these requirements is certainly essential to protect the privacy of users in a pervasive computing environment [see, also Myles et al. 2003; Warrior et al. 2003].

1.2 Different Categories of RBAC Constraints

In principle, RBAC supports the definition of arbitrary constraints on the different parts of an RBAC model [cf. Sandhu et al. 1996]. However, at first research efforts concerning RBAC constraints focused primarily on separation of duty constraints. With the increasing interest in RBAC in general and constraint-based RBAC in particular, research pertaining to other types of RBAC constraints also gained in importance [see, e.g., Bertino et al. 2001; Jaeger 1999]. In this paper, we especially deal with context constraints in RBAC environments. Subsequently, we describe some dimensions for the categorization of RBAC constraints that are relevant for the purposes of this paper. Then, we use these dimensions to explain our definition of context constraints. At first we differentiate between static and dynamic constraints:

—*Static constraints* are constraints that can be evaluated at "administration time" of an RBAC model, for example, static separation of duty (SSD) constraints which specify that two mutual exclusive roles must never be assigned to the same subject simultaneously.

—*Dynamic constraints* can only be checked at runtime according to the actual values of specific attributes or with respect to characteristics of the current session. For example dynamic separation of duty constraints which define that two mutual exclusive roles must never be activated simultaneously within the same user session, or time constraints which restrict role activation to a specific time interval (e.g., from 8 a.m. to 8 p.m.).

Another criterion to classify RBAC constraints is the distinction of endogenous (model intrinsic) and exogenous (environmental) factors:

—*Endogenous constraints* are constraints that completely relate to intrinsic properties of an RBAC model and inherently affect the structure and construction of a concrete instance of an RBAC model. For example, a static separation of duty (SSD) constraint on two mutual exclusive permissions prohibits an assignment of these permissions to the same role. Moreover, it also influences the definition of the respective role-hierarchy because it further prohibits that two distinct roles that possess the corresponding permissions can have a common senior role. Otherwise, a common senior could acquire both (mutual exclusive) permissions and thereby violate the corresponding SSD constraint [see, e.g., Ferraiolo et al. 1999; Strembeck 2004]. Depending on the respective implementation, similar effects can be observed for cardinality constraints for instance.

—*Exogenous constraints* are constraints that either exclusively involve attributes that do not belong to the core elements of an RBAC model (e.g., time constraints that restrict role activation to a specific time interval or allow access operations for a particular resource only on a specific weekday), or which refer to external (i.e., external to the RBAC model) attributes or properties of a specific RBAC model element (e.g., the location or current project assignment of a specific subject). In general, exogenous constraints are defined as conditions that take external data into account for certain operations or decisions of an access control service.

Beside the categorization as static/dynamic and endogenous/exogenous, constraints can also be subdivided in authorization constraints and assignment constraints:

—*Authorization constraints* are constraints that place additional controls on access control decisions. Thus, even if a subject is in possession of a permission that grants a certain access request, the access can only be allowed if the corresponding authorization constraints are fulfilled at the same time. For example, such constraints can be applied to implement access control policies based on access histories, as in Chinese Wall policies for instance.

—*Assignment constraints* are constraints that control the assignment or activation of permissions and roles (e.g., maximum and minimum cardinalities or separation of duty constraints). On the source code level, assignment constraints may be implemented through the same means as applied for authorization constraints (e.g., as an authorization constraint on the "assign role" or "activate role" permission). We think, however, that it is sensible to discriminate assignment and authorization constraints on the design level since both types address distinguishable intentions when engineering an RBAC policy.

The above categories are not completely orthogonal, and do not claim to provide a complete classification framework for all possible types of RBAC constraints. Nevertheless, these categories consider different aspects that can be observed individually, and facilitate the communication about RBAC constraints.

The remainder of this paper is structured as follows. In Section 2, we introduce the notion of *context constraints* as used in this paper. Subsequently, we describe an engineering process for the elicitation and specification of context constraints on the requirements level (Section 3). In Section 4, we then describe the conceptual structure of the xoRBAC component that can be implemented using any suitable programming environment. Especially we describe how context information, which is captured by special xoRBAC context functions, can be used to define context constraints. Afterwards, Section 5 shows how we used specific object-oriented techniques to implement a respective extension to xoRBAC and provide a language independent description of the functions that are needed to manage and enforce context constraints, before we discuss related work in Section 7. Section 8 concludes the paper.

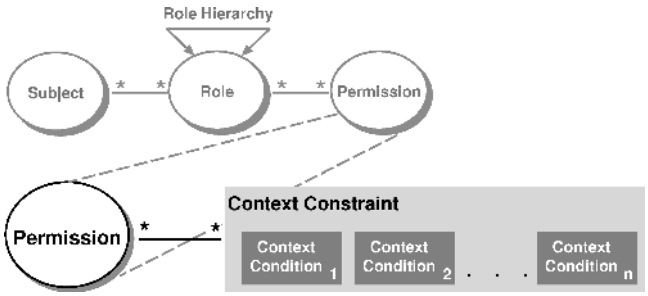

Fig. 1. RBAC permission with context constraint.

2. CONTEXT CONSTRAINTS

In the first place, a *context constraint* is an abstract concept on the modeling level (like other types of constraints, or the role concept are). A context constraint specifies that certain context attributes must meet certain conditions to permit a specific operation. With respect to the categories mentioned in Section 1.2, we thus define context constraints as *dynamic exogenous authorization* constraints. While context constraints can (in principle) also be applied as assignment or activation constraints (cf. Section 1.2), our hitherto experiences concerning the modeling and enforcement of context constraints are primarily based on the usage of context constraints as dynamic exogenous authorization constraints. As authorization decisions are based on the permissions a particular subject/role possesses, context constraints are associated with RBAC permissions (see Figure 1).

A context constraint is defined through the terms context attribute, context function, and context condition:

—A *context attribute* represents a certain property of the environment whose actual value might change dynamically (like time, date, or session-data for example) or which varies for different instances of the same abstract entity (e.g., location, ownership, birthday, or nationality). Thus, context attributes are a means to make (exogenous) context information explicit. On the programming level, each context attribute *CA* represents a variable that is associated with a $domain_{CA}$ which determines the type and range of values this attribute may take (e.g., date, real, integer, string).

—A *context function* is a mechanism to obtain the current value of a specific context attribute (i.e., to explicitly capture context information). For example, a function $date()$ could be defined to return the current date. Of course a context function can also receive one or more input parameters. For example, a function *age*(*subject*) may take the subject name out of the $\langle subject, operation, object\rangle$ triple to acquire the age of the subject, which initiated the current access request, e.g., the age can be read from some database.

—A *context condition* is a predicate (a Boolean function) that consists of an operator and two or more operands. The first operand always represents

```
% Definition of the access control function
check_access(Subject,Operation,Object) :-
    assigned_role(Subject,Role),
    has_permission(Role,Operation,Object),
    check_context_constraint(Subject,Operation,Object).
% Permission checking for roles and role hierarchies
has_permission(Role,Operation,Object) :-
    assigned_permission(Role,Operation,Object),
    permission(Operation,Object).
has_permission(Role,Operation,Object) :-
    super_role(Role,Super),
    has_permission(Super,Operation,Object).
% Evaluation of context constraints
check_context_constraint(Subject,Operation,Object) :-
    associated_cc(Operation,Object,CC),
    not violated(CC,Subject,Operation,Object).
```

Fig. 2. Excerpt from a Datalog specification.

a certain context attribute, while the other operands may be either context attributes or constant values. All variables must be ground before evaluation. Therefore, each context attribute is replaced with a constant value by using the corresponding context function prior to the evaluation of the respective condition.

—A *context constraint* is a clause containing one or more context conditions. It is satisfied iff (if and only if) all its context conditions hold.

The operator that is used in a context condition may be either a prefix operator that accepts two or more input parameters or a binary infix operator that compares two values (a left operand and a right operand). On the implementation level, it is of course possible to realize the functionality offered by context conditions in many different ways. An obvious option is to implement context functions, context conditions, and context constraints as separate entities, exactly as described above.

Context constraints are used to define conditional permissions. With respect to the terms defined above, a *conditional permission* is a permission that is associated with one or more context constraints and grants access iff each corresponding context constraint evaluates to "true". Therefore, conditional permissions grant an access operation iff the actual values of the context attributes captured from the environment fulfill the attached context constraints. The relation between context constraints and permissions is a many-to-many relation (see Figure 1). Thereby, a number of permissions can be associated with the same context constraint if necessary. Similarly, one permission may have associated with it many context constraints.

Figure 2 shows an excerpt of a logical definition of RBAC decisions in the presence of context constraints, written as a (stratified) Datalog specification [see, e.g., Apt et al. 1988]. The `check_access` predicate examines if an access request identified by the classical $\langle subject, operation, object \rangle$ triple can be granted or must be denied. The `assigned_role` and `has_permission` predicates detect the roles and permissions the subject possesses. The `check_context_constraint`

predicate determines the context constraints associated with a specific permission (an $\langle operation, object \rangle$ pair) and subsequently checks that none of these constraints is violated.

An algebraic definition of context constraints is given below (this definition extends the definitions provided by Ferraiolo et al. [2001], in particular the abbreviation PRMS refers to the set of permissions).

—ATTS, the set of context attributes (e.g., local_time, local_IP_address, subject_name, subject_age).

—DOMAINS, the set of available domains (e.g., boolean, date, integer, real, string).

—CONSTANTS $= \{x \mid x \text{ is a constant value} \wedge domain(x) \in \text{DOMAINS}\}$.

—OPERANDS $=$ ATTS $\cup$ CONSTANTS

—OPERATORS, the set of available (comparison) operators, for example, infix operators as $=, \geq, >, <, \leq, \neq$.

—$domain(oprtr : \text{OPERATORS}) \rightarrow \{d \subseteq \text{DOMAINS}\}$, a function to determine the set of domains an operator is specified for.

—$domain(oprnd : \text{OPERANDS}) \rightarrow \{d \in \text{DOMAINS}\}$, a function to determine the type of an operand.

—CONDITIONS $= 2^{\text{OPERANDS}} \times \text{OPERATORS}, \forall c \in \text{CONDITIONS} : c \mapsto \{(oprnd_1, \ldots, oprnd_x, oprtr) | oprnd_1, \ldots, opernd_x \in \text{OPERANDS}, oprtr \in \text{OPERATORS}\} \wedge \{domain(oprnd_1) \cup \cdots \cup domain(oprnd_x) \subseteq domain(oprtr)\}$.

—CC $= 2^{\text{CONDITIONS}}$, the set of context constraints.

—$conditions(cc : \text{CC}) \rightarrow \{cond \subseteq \text{CONDITIONS}\}$, a function to determine the conditions linked to a certain context constraint.

—PCL $\subseteq$ PRMS$\times$CC, a many-to-many permission to context constraint linkage relation.

—$linked\_ccs(p : \text{PRMS}) \rightarrow \{constraints \subseteq \text{CC}\}$, the linkage of a permission p to a set of context *constraints*. Formally: $linked\_ccs(p) = \{c \in \text{CC} \mid (p, c) \in \text{PCL}\}$

Endogenous constraints, as separation of duty constraints or cardinalities, for example, can often be derived from the business rules of a particular organization, e.g., constraints like: the roles "accounting clerk" and "controller" must be statically mutual exclusive, or the minimum user cardinality for the "controller" role is "one". In contrast to that it is, in our experiences, more complicated to specify exogenous (context) constraints. In Section 3, we therefore propose an engineering process for context constraints.

3. ELICITATION AND SPECIFICATION OF CONTEXT CONSTRAINTS

Context is an elusive concept, which has many different meanings to different people and communities. A definition for the meaning of *context* found in *Merriam-Webster's Collegiate Dictionary* is: "(1) the parts of a discourse that surround a word or passage and can throw light on its meaning (2) the interrelated conditions in which something exists or occurs."

In the area of *ubiquitous and pervasive computing* context can be defined as: "... any information that can be used to characterize the situations of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." [cf. Dey 2001].

In other words, context in general may consist of almost all information describing a specific situation. That is, context on the one hand consists of relatively static environment characteristics like a person's nationality, affiliation to an organization, or the salary of a certain employee. On the other hand, context also includes dynamic and often changing attributes like time, the location of a person or a device (physical and logical), proximity of other devices or proximity of a specific human being, history information stored in a log-file or database, the current CPU or network load, memory consumption of a specific device, and so on.

With respect to access control, one has to ask first which parts of these unmanageable quantities of context information are relevant for a specific authorization decision, and how the corresponding information may be elicited and defined on the modeling level. In this section, we therefore suggest a process for the elicitation and specification of context constraints. This process is based on goal-oriented requirements engineering techniques [see Antón 1996; van Lamsweerde 2001], and is designed as an extension to the scenario-driven role engineering process for RBAC roles presented in Neumann and Strembeck [2002]. Prior to describing the engineering of context constraints in detail, we give some background information concerning the scenario-driven role engineering process.

In the scenario-driven role engineering process usage, scenarios of an information system are used to derive permissions and to define tasks. In general, a *scenario* describes an action and event sequence, for example, to register a new patient in a hospital information system. Thus, each scenario consists of several steps, and a subject performing a scenario must possess all permissions that are needed to complete the different steps of this scenario. In turn, a *task* consists of one or more scenarios, and tasks are combined to form work profiles. A *work profile* comprises all tasks that a certain type of subject is allowed to perform. In a hospital environment different work profiles for physicians, nurses, and clerks are needed, for instance. In the role engineering process, work profiles are then used together with the permission catalog and the constraint catalog to define a concrete RBAC model. However, the scenario-driven approach presented in Neumann and Strembeck [2002] only provides general guidance for the subprocess of defining (exogenous) constraints. This fact and our aim to specify and enforce context constraints in an RBAC environment led us to the definition of the process extension proposed in this section.

3.1 Description of the Engineering Process

Figure 3 depicts an activity diagram for the engineering (sub)process. Like the role engineering process as a whole, the engineering of context constraints is in essence a requirements engineering process. To elicit context constraints we

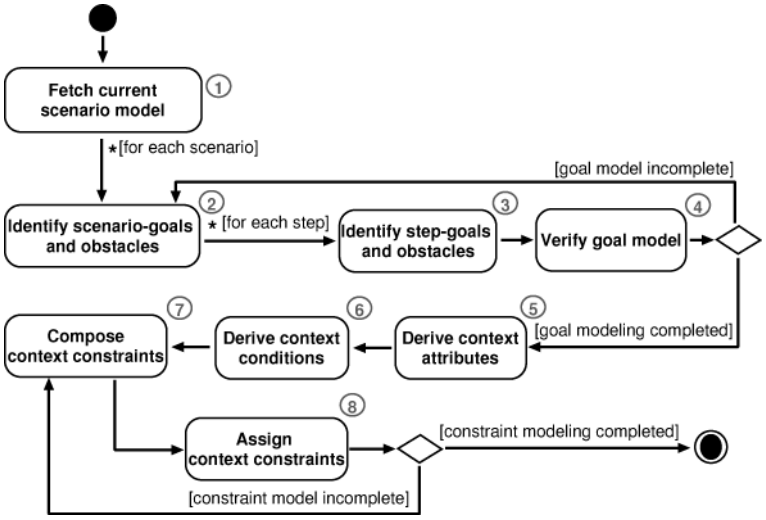

Fig. 3. A process for the elicitation and specification of context constraints.

especially use goals that are a familiar concept in the area of requirements engineering [see, e.g., Antón 1996; van Lamsweerde 2001]. Furthermore, goals are well suited to be applied in combination with scenarios to elicit and define requirements and to drive a requirements engineering process [see, e.g., Jarke et al. 1998; Rolland et al. 1999]. In general, a *goal* is an objective that the system under consideration should or must achieve. Goals can be defined on different levels of abstraction, ranging from high-level business goals to low-level technical concerns. Furthermore, goals may be used to represent functional as well as nonfunctional aspects like performance for instance. An *obstacle* is an undesired condition, which obstructs the fulfillment of one or more goals. Thus, obstacles can be seen as the opposite of goals. In the area of requirements engineering, obstacles are a valuable means to define more complete and more realistic requirements [see, e.g., van Lamsweerde and Letier 2000].

Scenarios and the scenario model serve as the basis for the scenario-driven role engineering process [Neumann and Strembeck 2002]. The first step of the constraint engineering subprocess shown in Figure 3 is thus to *fetch the current scenario model*. The succeeding activities are now described in more detail:

—*Identify scenario-goals and obstacles*: In this activity the goal(s) and obstacle(s) associated with each scenario are identified and explicitly modeled by filling out a small goal-template (obstacle-template), which consists of attributes like name, author, sub-goal-of, super-goal-of, and associated-with-scenario.

—*Identify step-goals and obstacles*: For each step within a scenario the associated goal(s) and obstacle(s) are identified and attached to the goal model. Each step-goal is a natural sub-goal of the corresponding scenario goal(s).

—*Verify goal model*: Here, the goal model produced in the preceding steps is verified and further elaborated. This activity is essential for the purpose

of defining stable goals that reflect the (security) demands on the system under consideration. To accomplish this task security engineers rely on the assistance of domain experts, for example, a bond dealer, an executive officer, and a clerk for a banking information system. The activities 2–4 are repeated until the goal model is completed (see Figure 3), that is, until the security engineers and domain experts define the model as adequate.

—*Derive context attributes*: Each goal (and obstacle) is examined to derive the context attributes that are needed to describe/fulfill this particular goal (e.g., daytime, a user's nationality, or the IP address of the host computer a specific service is requested from). Each context attribute is given a descriptive name, and explicitly stored together with a link to the goal(s) or obstacle(s) it has been derived from. Though it is often possible to straightforwardly derive context attributes and context conditions from goals (obstacles) in a single step, we model each as an own sub-activity to ensure that it is not omitted. In the further course of the process, context attributes are used to decide if a specific access control service is able to enforce context conditions based on a particular context attribute, for example, time information, or the access history of a particular subject (see Section 4).

—*Derive context conditions*: The goals and obstacles are now used to specify context conditions. Each goal and obstacle is a potential source of an access control relevant context condition and is thus analyzed individually. Since obstacles describe what should *not* happen, they are particularly useful in the derivation of context conditions. At this stage of the process (which is still focused on requirements engineering) we make no demands on the way context conditions are specified. For example, they can be defined as short sentences like "the IP address of the requesting computing device must have the value x", or "the request can only be granted if the requesting subject has already finished the processing of document b". However, the examples above can also be defined in a much shorter form like "IP address = x", and "access-history = document b", for instance. Each context condition is then stored together with a link to the originating goal/obstacle and scenario. Moreover, each context condition is classified if it can be enforced by the corresponding access control service, that is, if it can be mapped to the functions offered by a concrete access control service (see Section 4).

In our experiences, good reasons exist to model context conditions (and context constraints) even if they cannot (yet) be enforced on a technical level. The aim to specify and maintain a comprehensive, and preferably complete, access control policy for an information system is perhaps the most important reason. Such a "complete" policy rule set provides a valuable source of information for the corresponding security engineers. For example, it is then possible to identify which subset of an organization's access control policy rule set can (already) be enforced by the runtime system and which security goals can not be achieved yet. These information can be applied to thoroughly configure the respective access control service and to avoid security breaches that could result from unavailable information. Furthermore, a "complete" description of an access control policy rule set on the requirements level can

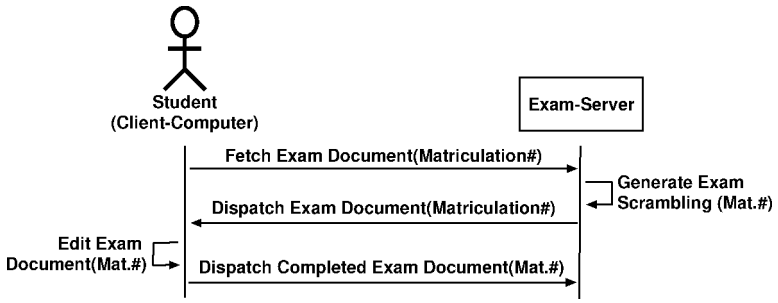

Fig. 4. A simple scenario for online examinations.

drive the technical evolution of access control services to close the gap between an abstract (complete) policy rule set and its enforceable subset.

—*Compose context constraints*: In this activity, context conditions are composed to form context constraints. Each context constraint comprises one or more context conditions. A context constraint that consists of two or more context conditions thereby defines each associated context condition must hold to fulfill this particular constraint. Context constraints are stored in a constraint catalog.

—*Assign context constraints*: Each constraint can be traced back to the context condition(s), control objective(s), obstacle(s), and/or scenario(s) it originates from—remember that we derive permissions from scenarios and compose work profiles of tasks/scenarios [cf. Neumann and Strembeck 2002]. Thus, we can identify the permission(s) a context constraint could sensibly be assigned to in a straightforward manner. The activities seven and eight are repeated until the constraint model is complete (see Figure 3), that is, until the security engineers define the model as adequate.

3.2 A Small Example

We now give an example for the engineering of context constraints as described in the previous section. Since a detailed case study would fill its own paper, we chose a simplified example that, however, provides additional insights into the process, and allows for an intuitive understanding of the corresponding activities.

Figure 4 shows a scenario for online examinations, depicted as message sequence chart. For example, online examinations are sensible for tests where students have to show their ability to use certain software tools (e.g., a programming language compiler, or a CASE tool), or for tests that should be analyzed (semi)automatically in a subsequent step.

In our example, a student first sends a "fetch" request for an exam document together with her matriculation number to the exam-server (for the sake of simplicity, we assume that a proper authentication procedure already took place, for example, by using a Kerberos-based mechanism). The exam-server then generates an individualized scrambling of the exercises to counteract

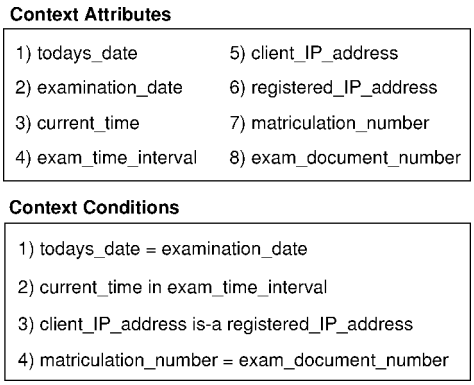

Fig. 5. Context attributes and conditions.

cheating or cooperation attempts of students (we presume, that all students take the exam within an invigilated PC pool). Afterwards, the exam document is dispatched to the client. Subsequently, the student edits the exam document and finally dispatches the completed exam document back to the server.

By applying the permission derivation procedure described in Neumann and Strembeck [2002], we can identify the following student permissions as ⟨operation, object⟩ pairs: ⟨fetch exam⟩, ⟨edit exam⟩, ⟨dispatch exam⟩. In other words, a student (resp, the student role) needs to be equipped with these permissions to successfully perform the scenario shown in Figure 4.

Subsequently, we conduct the engineering process for the elicitation and definition of context constraints as described in Section 3.1. This results in the following (condensed and simplified) goals and obstacles (we use a leading *G* for goals and a leading *O* for obstacles):

G_1 Enable online examinations.
- $G_{1.1}$ Provide an individual scrambling for each student.
- $G_{1.2}$ Ensure that students can edit their individual exam only.
- $G_{1.3}$ Ensure that students can fetch and dispatch their individual exam documents.
- $G_{1.4}$ Ensure that only registered PCs can be used to access the exam-server.
- $G_{1.5}$ Ensure that student access to the exam-server is limited to a specific date and a specific time interval.
- $O_{1.1}$ Student X can read or write the exam document of student Y.
- $O_{1.2}$ The exam-server can be accessed from an unregistered client PC.
- $O_{1.3}$ Student X is able to access the exam-server prior to, or after, the specified date and time interval.

According to the process shown in Figure 3, we now derive the context attributes and context conditions from the above goals and obstacles (see Figure 5).

Afterwards, the context conditions are used to compose context constraints, which are then assigned to permissions (see Figure 3). According to the above

goals and obstacles, we compose three context constraints and link them to the permissions derived from the scenario depicted in Figure 4. For the sake of simplicity, the context constraints are written as a list of conditions (cf. Figure 5) surrounded by curly brackets:

—⟨fetch exam⟩ {Cond_1, Cond_2, Cond_3}
—⟨edit exam⟩ {Cond_2, Cond_3, Cond_4}
—⟨dispatch exam⟩ {Cond_1, Cond_3, Cond_4}

Note that in the most simple case each context constraint consists of exactly one context condition. Context constraints composed of more than one condition are used to explicitly express the coherence and need for simultaneous validity of several conditions when performing a certain operation, that is, when using a specific permission.

The role and constraint engineering processes result in a concrete RBAC model (i.e., a concrete set of access control policy rules). The elements of this RBAC model are roles and role-hierarchies, permissions, and (context) constraints [see also Neumann and Strembeck 2002]. The xoRBAC software component [Neumann and Strembeck 2001] provides an RBAC service that (among other things) supports role-hierarchies, separation of duty constraints, and cardinality constraints for both roles and permissions. Nevertheless, to actually enforce RBAC policies that make use of context constraints on a technical level, a respective RBAC service must provide means to map modeling-level context constraints to concrete implementation structures. Sections 4 and 5 describe how we extended the xoRBAC component to enable the definition and enforcement of context constraints.

4. xoRBAC: CONCEPTUAL STRUCTURE

Figure 6 depicts the conceptual structure of the xoRBAC component. Permissions, roles, and subjects are the basic elements of xoRBAC. The *Subject Management* subcomponent provides means to manage subjects, that is, the entities that may actively initiate an operation. xoRBAC comprises static and dynamic constraint management as individual subsystems (see Figure 6). The *Static Constraint Management* of xoRBAC is based on permissions and roles and enables the definition of SSD constraints and cardinalities. The *Dynamic Constraint Management* allows for the definition of context conditions and context constraints (see Section 2).

Context constraints can be defined for "ordinary" access permissions as well as for administrative permissions such as assignment or revocation operations for example. The *Role Hierarchy Management* uses the static constraint management component to prevent the creation of role hierarchies that are disallowed by the SSD constraints or cardinalities within the system. The *Access Control Policy Management* additionally includes the decision component and the *Assignment Unit* for permission/role and user/role assignment and activation (see Figure 6).

The *Decision Component* contains the *Environment Mapping*, which captures context information via sensors, and the *Constraint Evaluation*, which checks

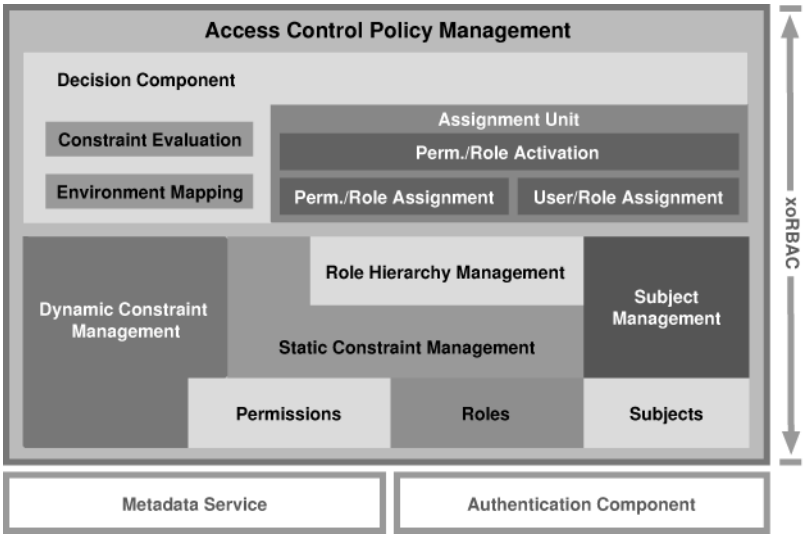

Fig. 6. Conceptual structure of the xoRBAC software component.

if the collected values match the context constraints associated with a certain conditional permission. Furthermore, xoRBAC is associated with a metadata service that records logging and audit information and enables the serialization (and recreation) of xoRBAC run-time instances by using XML encoded models as serialization format.

To reach an authorization decision for a particular access request, the decision component of xoRBAC receives the following information as parameters: the ID of an authenticated subject, that is, the user or user-agent who requests an access, the operation to be performed, and the name of the object which is the target of the operation—resulting in a $\langle subject, operation, object \rangle$ triple. If one ore more context constraints need to be checked to reach a certain access control decision, the decision component uses the corresponding functions provided by the environment mapping and constraint evaluation subcomponents (see Figure 6).

In essence, the *Environment Mapping* component comprises the *sensor library* of the xoRBAC access control service (see Figure 7). It manages all sensors connected to xoRBAC. Therefore, every sensor must be registered in the sensor library before it can be used within xoRBAC. Each sensor provides one or more context functions.

A context function is a mechanism to obtain actual values for specific context attributes (i.e., to explicitly capture context information). In other words, context functions are used to filter environment information and to make the current value of a relevant attribute available so that it can be used by xoRBAC (see also Section 2). Therefore, each context attribute that can be provided by a respective context function can be used to define context conditions (cf. Figure 7).

The sensor library of xoRBAC can be extended with arbitrary new sensors, respectively their corresponding software interface. This means that xoRBAC

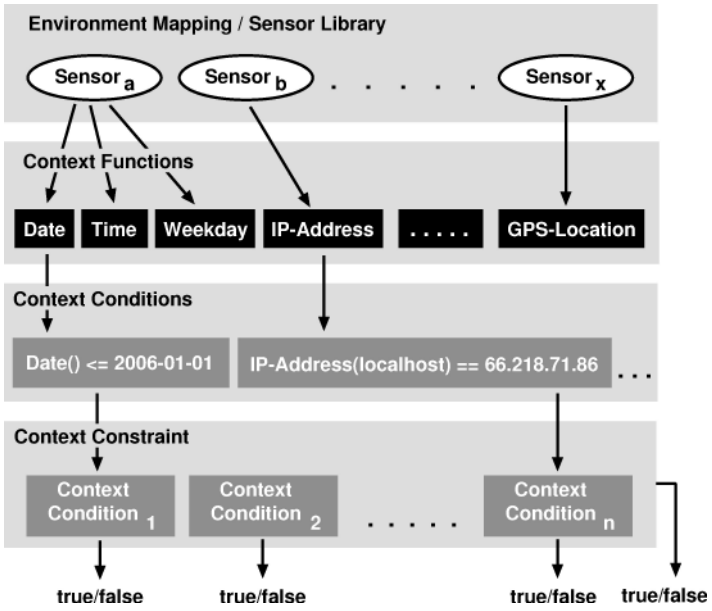

Fig. 7. From sensors to context constraints.

can be connected to hardware as well as software sensors to capture context attributes (see Section 5.5).

The *constraint evaluation* component checks if a set of actual sensor values match the corresponding context constraints and returns either `true` or `false` depending on the result of the evaluation. In this sense, context constraints provide sensor fusion, that is, they combine and interrelate the measurements of several sensors.

Thus, xoRBAC sensors and xoRBAC context constraints represent two different layers. A sensor (resp., the corresponding context functions) only captures "raw" context information from the environment and makes the respective context attribute available, for example, as string or numerical value. In turn, a context constraint uses the actual values of context attributes to check the associated context conditions and to decide if the corresponding access can be granted.

Figure 8 shows the definition process of concrete xoRBAC context conditions as activity diagram. For each modeling level context condition (see Section 3.1), we examine if the corresponding (abstract) context attribute(s) can be captured by an actual context function of an xoRBAC sensor. If so, a respective concrete context condition is specified (see Section 5.3). If, however, no appropriate context function is available, one may either implement a new context function or a new sensor in order to enforce the corresponding modeling level condition, or the corresponding modeling level condition is marked as "not yet enforceable". In our experiences, good reasons exist to model context conditions (and context constraints) even if they cannot (yet) be enforced on a technical level (cf. Section 3.1).

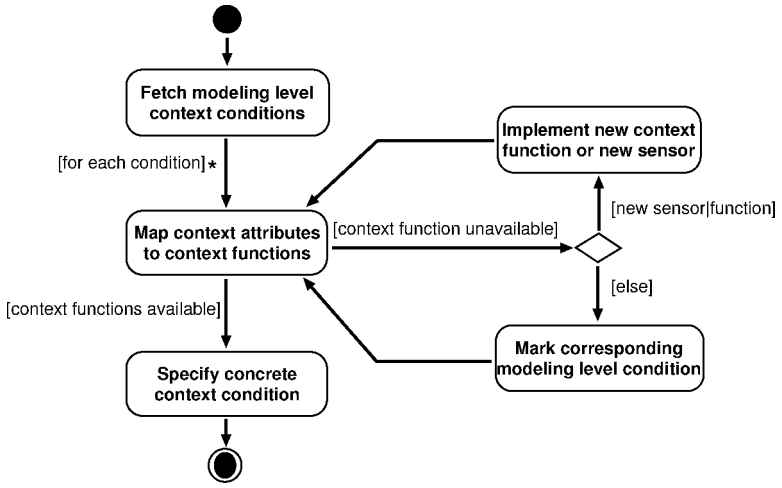

Fig. 8. Definition of concrete context conditions.

5. xoRBAC: IMPLEMENTATION

The xoRBAC access control component is a freely available software component implemented with XOTcl (eXtended Object Tcl) [Neumann and Zdun 2000]. XOTcl is a general-purpose object-oriented programming language that can be dynamically loaded into every Tcl compatible environment and is embeddable in C programs. As a Tcl extension, all Tcl commands [Ousterhout 1994] are directly accessible in XOTcl. XOTcl preserves the flexibility of Tcl and adds new language constructs to provide a highly flexible object-oriented programming environment.

In the implementation of xoRBAC, we especially used the dynamic object aggregation feature and the per-object-mixin language construct of XOTcl. *Dynamic object aggregation* enables the dynamic aggregation and disaggregation of objects at run-time. A *per-object mixin* (POM) is a class that is inserted at the beginning of the precedence order of a particular object. In other words, POMs are inserted in front of the precedence order induced by the class-hierarchy from which the object was instantiated. Thus, POMs are a means to extend every single object with additional behavior or capabilities dynamically at run-time [see Neumann and Strembeck 2001; Neumann and Zdun 2000]. However, as already mentioned, the design and abstract architecture of xoRBAC presented in this paper can of course be implemented using other programming languages as well.

Some important features of xoRBAC are: definition of arbitrary role-hierarchies (permission-inheritance), user-role review, user-permission review, permission-role review, definition of separation of duty constraints (constraint inheritance via the role hierarchy), and definition of maximum and minimum cardinalities (for further details see Neumann and Strembeck [2001]). In this section, we describe an extension of xoRBAC that enables the definition and enforcement of context constraints on permissions (see also Sections 2 and 4).

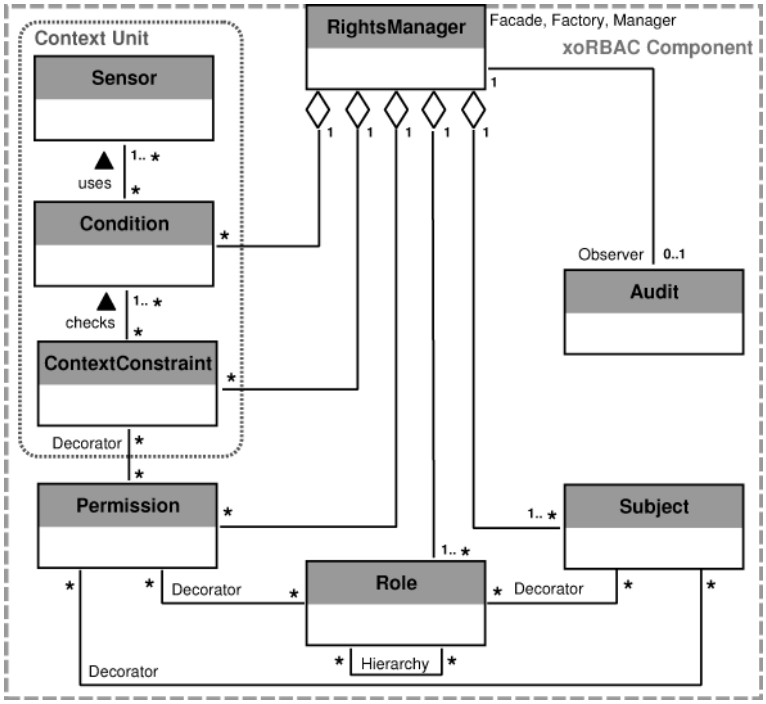

Fig. 9. Essential class relations of the xoRBAC component.

5.1 Static Design-Time Structures

Figure 9 depicts the essential design level class relations of the xoRBAC component. Several design patterns [see Gamma et al. 1995] are used in the implementation of xoRBAC. For example, the `RightsManager` class serves as *Facade* for the xoRBAC component, that is, it hides xoRBAC internal structures from other components that use xoRBAC. Thus, every external component uses xoRBAC through a well-defined API offered by the `RightsManager` class. At runtime, an `Audit` object can be registered for the `RightsManager` object according to the *Observer* pattern. The user-role assignment and the permission-role assignment relations are implemented using the *Decorator* pattern [for details see Neumann and Strembeck 2001].

As shown in Figure 9, the xoRBAC component basically consists of eight classes. The classes `Sensor`, `Condition`, and `ContextConstraint` form the *Context Unit* of xoRBAC. The context unit extends xoRBAC with functions that allow for the specification and enforcement of context constraints as described in Section 4.

The `ContextConstraint` class is defined as a meta-class, which means that its instances are regular classes [for further information on XOTcl meta-classes see also Neumann and Zdun 2000]. Thus, for each conceptual context constraint that was defined during the engineering process (see Section 3) a respective `ContextConstraint` instance is created. At run-time, each of these instances checks *exactly one* (modeling level) context constraint. Further on, each actual

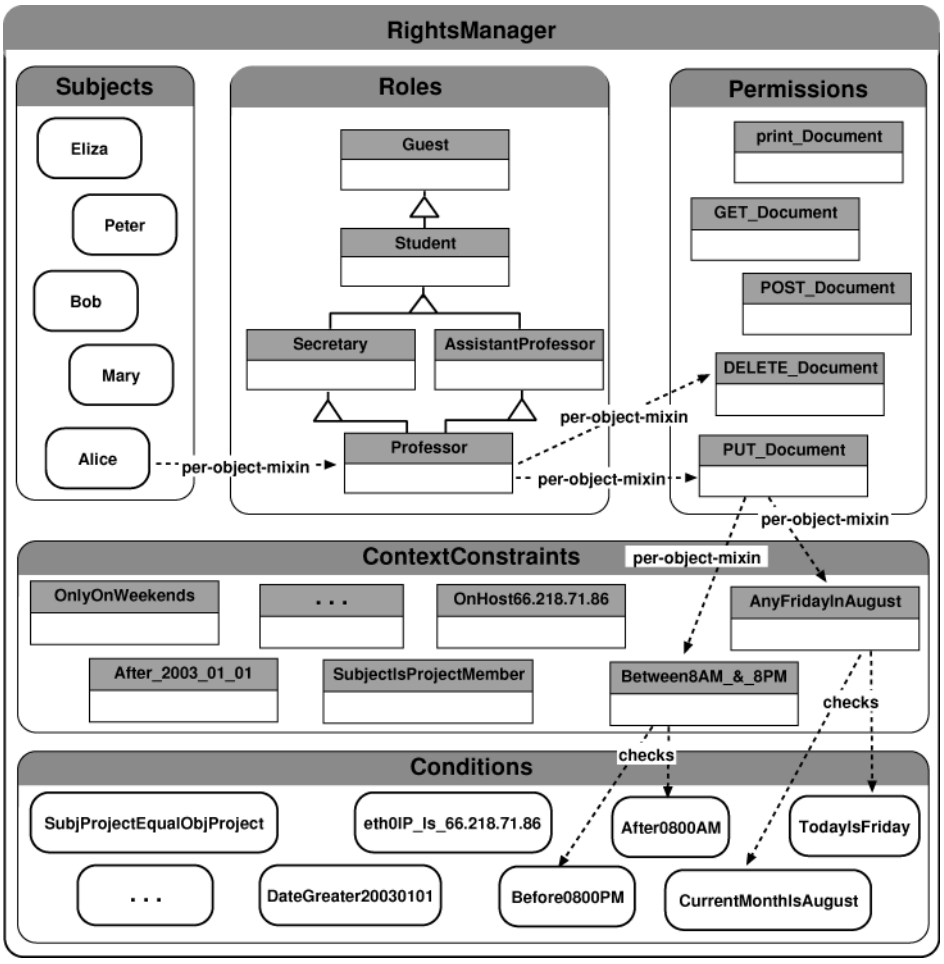

Fig. 10. A `RightsManager` object at runtime.

`ContextConstraint` checks one or more `Condition` objects. And each `Condition` object uses either one or more `Sensor` objects to implement a specific modeling level context condition.

5.2 Dynamic Run-Time Structures

In xoRBAC, new sensors, conditions and context constraints (i.e., instances of the `Sensor`, `Condition`, and `ContextConstraint` classes) can be dynamically defined. In other words, the sensor-library and the pool of context conditions and constraints can be dynamically extended.

Figure 10 depicts a `RightsManager` object at run-time, it shows the dynamic object aggregation of `Subject`, `Role`, `Permission`, `ContextConstraint`, and `Condition` instances, and their encapsulation within a respective namespace. In xoRBAC, POMs are used to assign roles to subjects and to assign permissions to roles [Neumann and Strembeck 2001]. In the same way, xoRBAC uses POMs

Table I. Functions Needed for Access Decisions

Function	Semantics
check_access(s,op,ob)	Check if subject s is allowed to perform operation op on object ob. The function (implicitly) performs a role and permission lookup and checks respective context constraints if necessary (via the `not_violated` function).
not_violated(cc)	Check if all conditions linked to context constraint cc are satisfied. For this purpose, the `not_violated` function invokes the satisfied function for all associated conditions. The function returns `true` if all conditions are satisfied and `false` otherwise.
satisfied(c)	Check if the predicate defined by condition c is satisfied. For this purpose, the satisfied function invokes the `evaluate_predicate` function with the corresponding operator and operands as input parameter.
evaluate_predicate(oprtr,oprnds)	Evaluate the predicate consisting of the operator oprtr and a list of operands given by the oprnds parameter. Each operand may either be a constant value or a variable representing a context attribute. Since all variables must be ground before evaluation, the `evaluate_predicate` function uses the respective context_function functions to receive the current values of relevant context attributes.
context_function(ca)	A `context_function` captures the value of a specific context attribute ca in the exact moment the respective function is called.

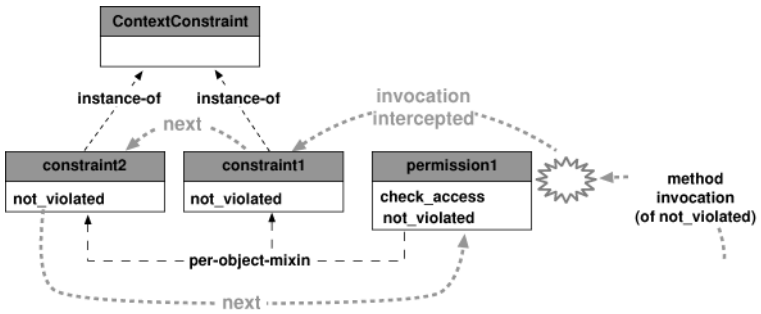

Fig. 11. Next-path for the call of `not_violated`.

to associate permissions with context constraints. The `ContextConstraints` are linked to `Permission` objects according to the *Decorator* pattern (see Figure 9). The use of POMs allows to dynamically (de)register `ContextConstraints` for `Permission` objects at arbitrary times.

For each `ContextConstraint` the `not_violated` function checks if all `Condition` objects that are registered for this particular constraint are satisfied (see also Table I). Figure 11 depicts two context constraints `constraint1` and `constraint2`, which are registered as POMs for `permission1`. Each method call to `permission1` is intercepted and redirected to its POMs `constraint1` and `constraint2` prior to invoking the respective method in `permission1`. This feature is called method combination or *method chaining* [see also Neumann and Zdun 2000] and is a well-known approach to handle dynamic class structures.

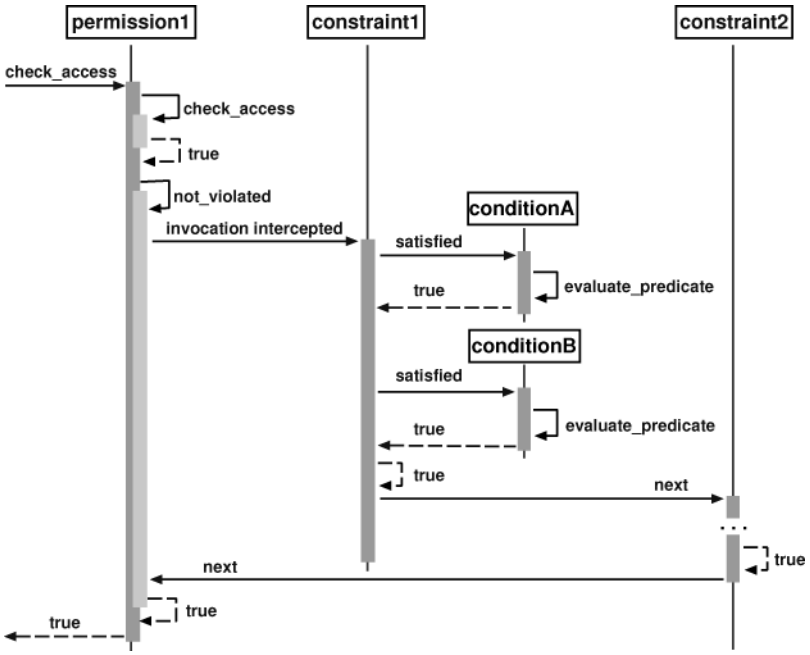

Fig. 12. Message sequence chart of a `not_violated` call for the return of `true`.

In particular, Figure 11 shows a simple example of a next-path resulting from a call of the `not_violated` function on `permission1`. The `not_violated` call to `permission1` is intercepted and passed along the next-path to its POMs `constraint1` and `constraint2` and finally back to `permission1`. Regarding the `not_violated` function, the permissions and context constraints of xoRBAC form a *Chain of Responsibility* [see Gamma et al. 1995]. This means, a `not_violated` call is passed along the next-path until a context constraint is violated and denies the request by returning *false*. However, if all context constraints return *true* the call is finally passed back to `permission1`, which then grants the corresponding access request.

Figure 12 shows a message sequence chart of a `not_violated` call for the return of `true`. Section 5.4 provides a detailed description for access control decisions with conditional permissions in xoRBAC.

5.3 Specification of Context Constraints

Each `Condition` object implements one particular context condition (see Section 2). In essence, a `Condition` object consists of an *operator* and a number of *operands*. At least one operand is always represented by a particular context function, which captures the current value of a specific context attribute. The other operands may be either constant values or other context attributes (resp., context functions). The operator is used to compare the operands. This means, a `Condition` object either compares the results of two (or more) context functions, or the result of one (or more) context function and a number of constant values. For this purpose the `satisfied` function is called (see Table I).

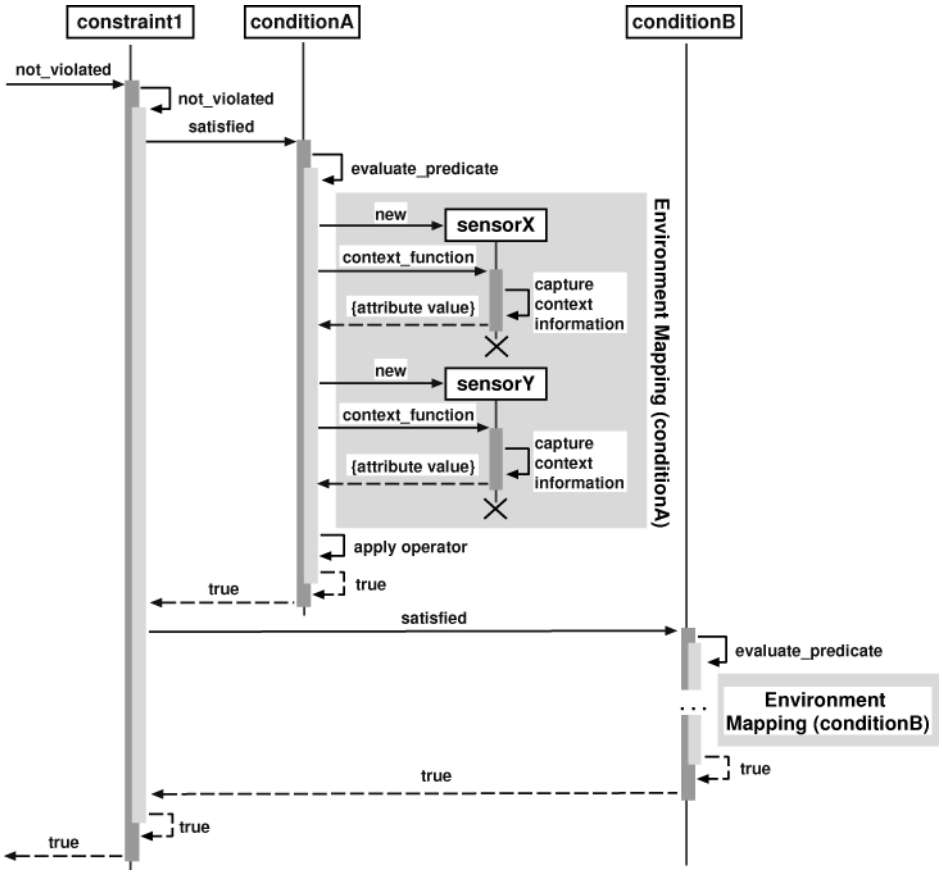

Fig. 13. Evaluation of a context constraint at run-time.

Because each context condition represents a predicate, a concrete `Condition` object returns either true or false as result of a `satisfied` call (see also Section 2). In particular, the `satisfied` function uses the `evaluate_predicate` function (see Figure 12 and Table I).

When a predicate script is evaluated at run-time it first instantiates the `Sensor` objects needed to check the corresponding condition (see also Figure 13). Thus, in xoRBAC every `Condition` object employs its own volatile `Sensor` objects to capture a consistent snapshot of the relevant context attributes (cf. Section 4). Next, the corresponding context functions are executed (all variables must be ground before evaluation—see Section 2), and the respective results are compared using the corresponding comparison operator. Each xoRBAC `Condition` object contains a predicate script. *Predicate scripts* are automatically generated from the operator and operands defined for a particular condition. A predicate script delivers either a return value of "1" (true), or "0" (false).

To conveniently manage xoRBAC, we developed a graphical front end that allows for the administration of xoRBAC run-time instances. Moreover, we developed a graphical tool that provides support for the scenario-driven role

Table II. Context Constraint Specific Functions

Function	Semantics
create_condition(c,oprtr,oprnds)	Create a new condition c containing the operator oprtr and a list of operands oprnds.
delete_condition(c)	Remove all links between condition c and associated context constraints. Subsequently, delete condition c.
create_context_constraint(cc)	Create a new context constraint cc.
delete_context_constraint(cc)	Remove all links between context constraint cc and associated permissions. Subsequently, delete the context constraint cc.
link_condition(c,cc)	Link condition c to context constraint cc. Subsequently, c is checked each time cc is evaluated.
unlink_condition(c,cc)	Remove the link between condition c and context constraint cc. Subsequently, condition c is *not* checked when cc is evaluated.
link_constraint(cc,p)	Link context constraint cc to permission p. Subsequently, cc is evaluated each time a subject issues an access request associated with p.
unlink_constraint(cc,p)	Remove the link between context constraint cc and permission p. Subsequently, constraint cc is *not* evaluated when permission p is checked.
conditions(cc)	Return a list of all conditions currently linked to context constraint cc.
context_constraints(p)	Return the list of all context constraints linked to the permission object p.

engineering process and the constraint engineering process presented in Section 3.1. Nevertheless, xoRBAC can also be controlled "directly" via its API. Table II shows the list of context constraint specific functions.

5.4 Access Control Decisions

The access control function of xoRBAC is implemented via the `check_access` function, which receives the traditional access control triple $\langle subject, operation, object \rangle$ as input parameters. Thereby, xoRBAC provides a clear interface to other components that use xoRBAC as their access control service.

In xoRBAC permissions are always positive, that is, a permission always grants a certain access right and does not deny it. The processing of "ordinary" access requests is explicitly described in Neumann and Strembeck [2001]. Therefore, the focus of section is on access control decisions with conditional permissions. Figure 13 depicts a message sequence chart of an action and event sequence, which occurs if an access request is granted by a context constraint. Here `constraint1` is an instance of the `ContextConstraint` class, `conditionA` and `conditionB` are instances of the `Condition` class, and `sensorX` and `sensorY` are actual `Sensor` objects (see Figure 9).

Initially, `constraint1` receives a `not_violated` call (see Figures 12 and 13). Next, `constraint1` calls the `satisfied` function of all `Condition` objects associated with `constraint1`. At first the `satisfied` function of `conditionA` is called. Then, `conditionA` evaluates its predicate script to decide if this particular context condition is fulfilled. During the evaluation of the predicate

script, conditionA draws a snapshot of the relevant context attributes by using two (volatile) sensor objects sensorX and sensorY (note that the call of context_function in Figure 13 serves as a placeholder for any call of an actual context function—see also Sections 4 and 5.5). After conditionA has drawn a snapshot of the relevant context attributes it compares the obtained values by applying the corresponding comparison operator and returns either true or false according to the result of this comparison (see also Section 2). In Figure 13, the satisfied function of conditionA returns true. The same procedure is repeated for conditionB. Here, the satisfied function of conditionB also results in a return value of true. Because all context conditions associated with constraint1 are satisfied, constraint1 finally returns true as result of the not_violated call.

Note that the not_violated function of a ContextConstraint returns true iff each Condition object that is associated with this ContextConstraint is satisfied. Moreover, for a call of not_violated the corresponding Permission object is always the last object in the next-path and thus the last object within the chain of responsibility (cf. Section 5.2). This means, if no ContextConstraint previously denies the requested access by returning false the not_violated call is finally passed back to the respective Permission object, which then returns true to indicate that the corresponding access request can be granted (cf. Figure 12). Table I describes the functions that are needed for access decisions.

5.5 Sensor Library

We differentiate sensors in two coarse-grained categories: *hardware sensors* (i.e., pieces of hardware that can be accessed via a software interface), which capture information on a host's physical environment (e.g., current GPS location, temperature, noise-level, light, or proximity of an other device), and *software sensors*, which exclusively consist of software components and are used to gather information that can be extracted from system internal sources (e.g., the IP-address of a certain device, information stored in databases or log-files, the status of other applications or services, CPU ID, CPU state, network load, and so on).

In principle, both sensor types can be used to capture access control relevant context information. However, for the time being, we concentrate especially on the use of software sensors in xoRBAC. The use of software sensors is sensible for the purpose of access control because relevant access control related information is often stored through software-based services. For example, information like birthday, nationality, ownership, or physician to patient relations, can be gathered from specific databases or documents as birth certificates, contracts, passports, or patient records. Therefore (for the time being), it is convenient to read such information directly from the respective electronic sources by using software sensors, for example, through a context function that executes a certain database query, or a context function that reads a specific information from XML documents. Likewise, most system internal attributes can be conveniently captured via software sensors that query the status of a certain software service or look for specific log-file entries for example.

Table III. Sensor-Specific Functions

Function	Semantics
register_sensor(s)	Register sensor s in the sensor library. A sensor must be registered in the sensor library before any of its context functions (and thereby the resp. context attributes) could be used to define context conditions.
deregister_sensor(s)	Remove sensor s from the sensor library. After removing a sensor from the sensor library none of its context functions (and thereby none of the resp. context attributes) could be used define context conditions anymore. A sensor may only be deregistered if no condition currently uses the respective context functions.
announce_context_attribute(s,cf,ca)	Export the context function cf provided by sensor s to make context attribute ca available. Subsequently, ca can be used as an operand in the definition of context conditions.
withdraw_context_attribute(ca)	Withdraw context attribute ca from the list of available context attributes and deregister the corresponding context function. Subsequently, ca may no longer be used to define context conditions. A context attribute ca may only be withdrawn if it is not used in a context condition.

In general, each context attribute that can be captured by a context function can be used as an operand for the specification of xoRBAC context conditions (see Section 4). The different sensors and their context functions thus provide the "operand-vocabulary" of the xoRBAC component. Any sensor in xoRBAC can be extended with additional functions, and new sensors can be defined and registered at run-time. Table III describes the list of sensor-specific functions.

The xoRBAC component can be used for applications on Unix or Windows with a C or Tcl linkage [see also Neumann and Strembeck 2001]. However, some sensors may access platform-specific system functions, for example, to read the local-host's IP-address. Therefore, we install different sensor libraries depending on the platform xoRBAC is used on. Nevertheless, sensor interfaces are (of course) platform independent. This means that two sensors which provide the same function on different platforms offer the same interface but may access different implementations. A sensor's interface thus hides the implementation details from the xoRBAC component. Thereby, the sensor functions of xoRBAC are platform independent.

All sensors of xoRBAC are *passive sensors*. That means that the corresponding context functions do not permanently provide xoRBAC with topical context attribute values but are selectively polled to provide a "snapshot" of the context attributes that are needed for a particular authorization decision.

6. THE GRAPHICAL ADMINISTRATION TOOL

While xoRBAC can be completely controlled via its application programming interface (API), it also provides a tailored graphical administration tool that supports the complete set of functions offered by xoRBAC. The xoRBAC

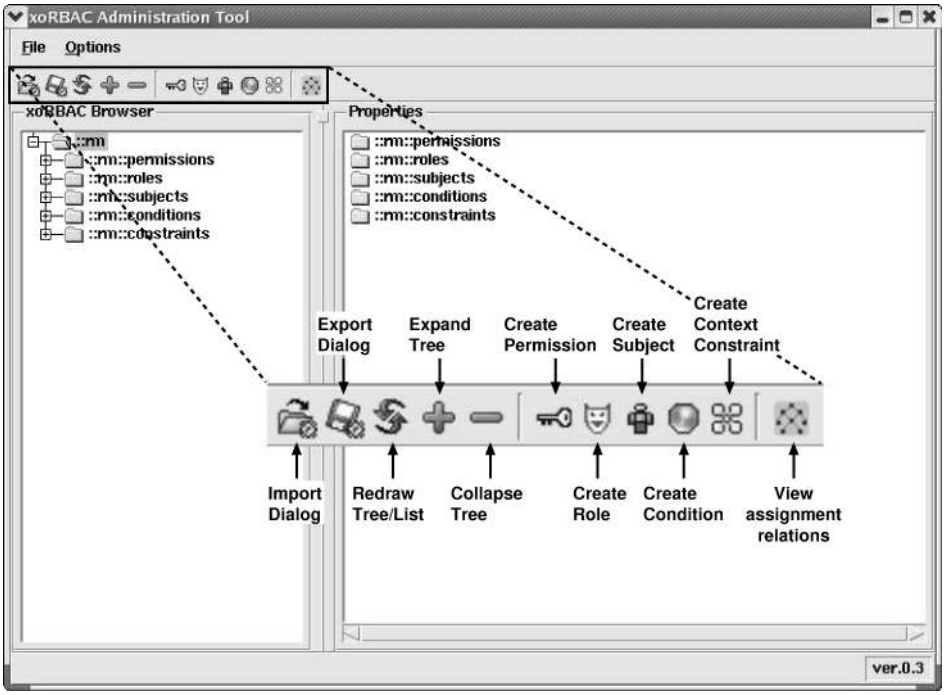

Fig. 14. Screenshot of the xoRBAC administration tool main window.

administration tool can be used to create roles, permissions, subjects, context conditions and context constraints, and to define and maintain the corresponding assignment relations. For example, it offers the following features:

—Many-to-many user-role and permission-role assignment (and revocation).
—Definition of arbitrary (DAG) role-hierarchies (permission-inheritance, see also [Neumann and Strembeck 2001]).
—Definition of context conditions, context constraints and conditional permissions.
—Definition of separation of duty constraints for both roles and permissions (SOD constraint-inheritance via the role-hierarchy, see also [Strembeck 2004]).
—Maximum and minimum cardinalities for both roles and permissions.
—Extensive review functions (introspection), e.g., subject-role review, permission-role review, subject-permission review, and graphical inspection of the corresponding relations.

The xoRBAC administration tool is a separate software component that is included in the xoRBAC package. It uses the xoRBAC API to access the different functions of xoRBAC. The administration tool is written in XOTcl and uses freely available widget sets. Figure 14 shows the main window of the xoRBAC administration tool, and the foreground of this figure depicts and describes the enlarged toolbar located at the top of the window.

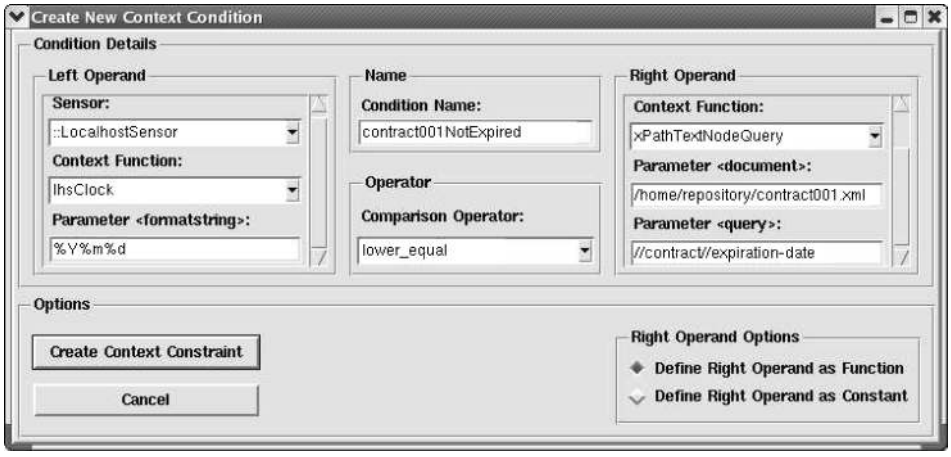

Fig. 15. Dialog for the creation of context conditions.

With respect to the main topic of this article we now take a closer look at the creation (and maintenance) of context conditions and context constraints (see Section 2). Figure 15 shows a screenshot of the dialog for the creation of context conditions. In particular, the screenshot depicts the specification of a context condition using a binary infix operator (here: `lower_equal`, $\leq$). The corresponding left operand is specified on the left-hand side of the creation dialog. In this example, the respective operand is "today's date", which is determined at run-time via the context function `lhsClock` provided by the `LocalhostSensor` of xoRBAC. The `lhsClock` context function receives a format string, which allows to parameterize the return value of this function. The `%Y%m%d` format string given in the example causes the `lhsClock` function to return the current date in a "YYYY MM DD" format.

The right-hand side of the dialog specifies the right operand. In Figure 15 this operand represents the expiration date of an XML-based digital contract. To read the expiration date from the corresponding XML document [Bray et al. 2004] we use the `xPathTextNodeQuery` context function provided by the `GenericXPathSensor` of xoRBAC (note that the combo box widget for sensor selection of the right operand is not visible in Figure 15 because we moved the scrollbar down to show the two parameters passed to the `xPathTextNodeQuery` function). The `xPathTextNodeQuery` context function receives two parameters defining an XML document (here: `/home/repository/contract001.xml`) and the corresponding XPath query (here: `//contract//expiration-date`). If the `contract001NotExpired` condition should be evaluated at run-time, the `lhsClock` function first determines today's date and the `xPathTextNodeQuery` function performs an XPath query [Clark and DeRose 1999] to read the value of the `expiration-date` element in the `contract001.xml` document (remember that all variables must be ground before evaluation, see also Sections 2 and 5.4). Subsequently, the condition is evaluated using the respective operator (here: `lower_equal`) and returns either `true` or `false`.

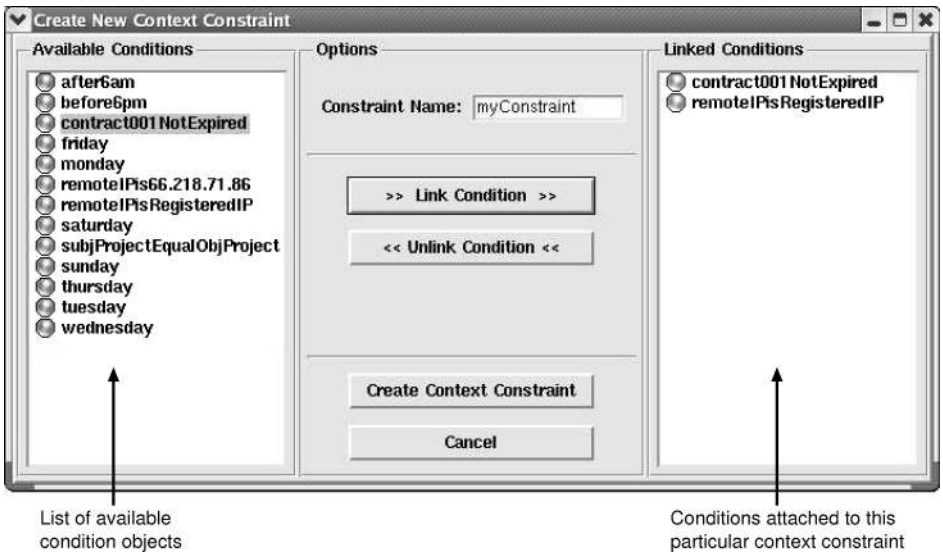

Fig. 16. Dialog for the creation of context constraints.

The context condition dialog shown in Figure 15 is dynamic to a certain degree. For example, the options/entries offered by the "Context Function" combo boxes depend on the sensor selected via the "Sensor" combo box. Moreover, the number of entry widgets allowing to pass parameters to the respective context functions is dynamically determined. This means that, depending on the chosen context function, the dialog draws the exact number of entry widgets (and corresponding captions) that are needed to specify parameter values for the respective context function. In Figure 15, for instance, we have one entry widget to pass a format string to the `lhsClock` function and two entry widgets to define a document and an XPath query for the the `xPathTextNodeQuery` function.

Having specified the context conditions, we can define context constraints (which consist of context conditions, see Section 2). Figure 16 depicts the dialog for the creation of context constraints. On the left-hand side we have the list of available condition objects. Each of these conditions may be linked to the corresponding context constraint object (here: `myConstraint`). And any condition object that is linked to a specific context constraint is evaluated each time the respective context constraint is evaluated (see also Section 5.4).

Subsequent to the creation of a context condition or context constraint, the dialogs shown above can be used to modify/maintain the respective condition and constraint objects and their relations.

7. RELATED WORK

A number of recent research contributions deals with different approaches to consider context information in access control decisions. These contributions range from abstract high-level models to case studies and concrete (implemented) software systems.

Adam et al. [2002] introduced a sophisticated authorization model that was specifically designed to meet access control requirements of digital libraries. In particular, their model allows for the consideration of additional user and object attributes aside from unique identifiers. Here, credentials represent attributes that describe certain characteristics and qualifications of users, like age, salary, nationality, or current project involvement for example. Likewise, digital library objects are associated with attributes describing the contents of these objects (e.g., via categories as taxation, civil law, information system research). Moreover, digital library objects are structured in different segments, like author information, abstract, sections, bibliography. These information are used to define fine-grained access control policies. Adam et al. [2002] implemented a prototype system that provides nearly all functions of their authorization model.

Various contributions concerning access control in collaborative environments exist, especially, for groupware and workflow systems. For example, Thomas and Sandhu [1997] introduced TBAC, a family of models that support the specification of active security models. In TBAC permissions are actively (de)activated according to the current task/process-state. Nitsche et al. [1998] gives a high-level description of a system extension that was implemented to consider context information for authorization decisions in medical workflows. Wolf and Schneider [2003] suggests to consider the authentication method that was applied to identify a particular subject (e.g., password-based versus certificate-based authentication) as context information to assign/activate certain roles. In Kang et al. [2001], an approach for access control in interorganizational workflows is suggested, and Bertino et al. [1999] presents a well-elaborated language and algorithms to express and enforce constraints, which ensure that all tasks within a workflow are performed by predefined users/roles. A more general language that allows for the specification of different access control policies that can coexist in the same system is presented by Jajodia et al. [2001]. One similarity for all of these approaches is that they allow to use some type of context information, for example, the execution history of individuals/roles and the current task, to make assignment, activation, or authorization decisions.

Georgiadis et al. [2001] introduce the context-based team access control model (C-TMAC) as an extension of the TMAC approach presented by Thomas [1997]. Here, a team is defined as a group of users acting in different roles with the objective of corporately completing a certain task, for example, a group of physicians and nurses attending a patient. Thus, in C-TMAC the team concept is used to associate users with contexts, like roles are used to associate users with permissions.

The problem of information sharing and security in dynamic coalitions [see Cohen et al. 2002; Phillips et al. 2002] is related to C-TMAC. A (dynamic) coalition consists of two or more different organizations (resp., their employees) that temporary work together to achieve a common goal. Coalitions can be formed dynamically, and each coalition member may share a number of information resources with other coalition members. However, each party must be able to individually tailor the access rules for their content according to the status of

other coalition members. Bharadwaj and Baras [2003] describes a preliminary approach to enable the automated negotiation of RBAC policies in dynamic coalitions.

El Kalam et al. [2003] introduced an access control model they call "organization-based access control". In particular, they introduce "organization" as a modeling level concept to group subjects. A subject may be either a user or an organization, and each subject can be assigned to a role. Moreover, they use a modeling level concept called "context". Here, a specific context is defined by a certain condition (e.g., "subject is attending physician", or "case of emergency"). The relation "permission" is then used to link organizations, roles, views, activities, and contexts ("organization v grants role w permission to perform activity x on view y in context z"). A more concrete example of a "permission" relation could be: "hospital x grants role Intern permission to read patient records in case of emergency".

Wang [1999] presents an approach to realize context- and role-based access control for a hypermedia environment. He uses three different role categories: roles that represent the job position of a user (e.g., "software engineer"), team roles to express team membership (e.g., "software testing group"), and personal roles that are assigned to individuals but may, in exceptional cases, be delegated to other persons in order to transfer individual job responsibilities. While subjects are managed via roles, hypermedia objects are managed using so called wrappers, i.e. wrappers serve as containers for several hypermedia objects. Context information is only implicitly included through the notions of team membership and process state (which can be derived from the wrapper a certain object is actually contained in).

Edjlali et al. [1998] presented a history-based access control mechanism for mobile Java code, called Deeds. They propose to utilize the access history of individual (esp., mobile) programs as context information to protect a host computer, respectively, information stored on the host, from potentially insecure/dangerous operation sequences. Jaeger et al. [1999] introduces a system architecture for the control of downloaded executable content. The underlying model was built to support both, system- and application-specific access control policies. In other words, administrators are able to define system wide mandatory access control policies, while individual users may perform additional discretionary access controls for specific subdomains (within the limits of the system wide policy). As context information Jaeger et al. particularly use the identity of the content provider and of the content itself, the identity of the downloading principal, and the actual state of the associated application. They implemented their security architecture based on IBM's Lava operating system environment.

Barkley et al. [1999] suggests a model to include relationships between real-world entities into RBAC access decisions. In particular, they propose to apply the resource access decision facility (RAD), defined by the Object Management Group (OMG), to combine the access decisions of two (or more) access control services. Their approach combines an RBAC service and a service that evaluates the relationships between real-world subjects and/or objects.

Wilikens et al. [2002] presented an approach that aims at context-related access control in the healthcare domain. They give an example that uses time constraints and the (physical) location of physicians and members of the nursing staff as context information to control access operations on patient records. Longstaff et al. [2000] presents an other approach from the healthcare domain which allows to override permission assignments (and even separation of duty constraints) in case of emergencies. However, a system which allows that specific policy rules or constraints are overridden must perform audits on the use of overrides to prevent/detect fraudulent use of this feature.

Role templates, as proposed by Giuri and Iglio [1997], could be used to consider certain types of context information. In particular, Giuri and Iglio suggest to parameterize roles and permissions to gain more flexibility (and less redundancy) compared to treating them as fixed entities. For example, instead of defining an own role-hierarchy for each project within an organization a generic parameterized hierarchy may be defined. Users are assigned to a specific project-role, and the name of the concrete project or department is used as a parameter for the assignment operation. Hence, according to the concrete parameter value, a user may only access resources that are allocated to her project or department.

Covington et al. [2001] described an approach which uses two different kinds of roles to assign rights to users and to include context information in an Intelligent/Aware Home environment. They suggest to use classical RBAC roles to provide subjects with permissions. Besides, they introduce the notion of "environment roles" that are automatically (de)activated by the environment (the aware home) to depict the actual environmental context. In essence, environment roles are bound to environment conditions that can be captured by the (hardware) sensors within the Aware Home, like time, the day of the week, room temperature, or location of a user. Environment roles are activated according to these conditions and are used together with subject roles to reach an authorization decision.

The TRBAC model, presented by Bertino et al. [2001], allows for the periodic (de)activation of roles and for the definition of temporal dependencies among the actions that (de)activate roles. In particular, they use active rules, so called role triggers, to define temporal dependencies between activation events. Thus, time in general and time intervals between activation events in specific are used as context information in TRBAC. Atluri and Gal [2002] presents a formal authorization model for temporal data. Their model is especially focused on access control measures for web-based information portals.

Yao et al. [2001] described the support of active security in the OASIS role-based access control architecture. In OASIS, role activation is governed by rules that are specified in logic. The corresponding rules may also specify certain preconditions that must be fulfilled to activate a particular role. Moreover, these conditions are bound to events that cause that a role is deactivated as soon as a condition becomes false. Likewise, the rules specifying access to objects or services can be bound to conditions/attributes that need to be evaluated each time a rule is applied. The time of day, or user attributes, like membership in

a certain group, are examples for such environmental attributes. Bacon et al. [2001] presents an approach to express OASIS policy rules, for example, for role activation, in pseudo-natural language statements that can be translated into first-order logic with side conditions.

Belokosztolszki et al. [2003] presented an approach to control information flow in (and out of) the OASIS RBAC system. In particular, they use so-called "contexts" to classify elements of their RBAC system. These contexts are applied to control information flows between system entities. Note that the notion of context used by Belokosztolszki et al. does *not* refer to dynamic environment parameters. In fact, in Belokosztolszki et al. [2003] a "context" roughly corresponds to "security labels" in lattice-based access control [Denning 1976].

McDaniel gives a good overview on the consideration of context in authorization policies (on a technical level). Moreover, he presents the Antigone Condition Framework (ACF), which supports the creation and evaluation of policy conditions [McDaniel 2003]. ACF is intended to be a general-purpose condition framework, which can be applied to extend other services with ACF conditions. In essence, ACF conditions are (parametrized) Boolean functions. ACF conditions are similar to *context conditions* as defined in Section 2.

8. CONCLUSION AND FUTURE WORK

This paper introduced a framework for a special kind of RBAC constraints, called context constraints, which are defined as dynamic exogenous authorization constraints. We specified the required terminology and provided a definition for context constraints. We presented an elicitation process to derive context constraints during role engineering and described an implementation that extends an existing RBAC system to enable the enforcement of context constraints.

The presented process for the elicitation and specification of context constraints is based on goal-oriented requirements engineering techniques. This process is designed as an extension to the scenario-driven role engineering process for RBAC roles [Neumann and Strembeck 2002]. The overall process provides guidance for security engineers and allows for the specification of concrete RBAC models including context constraints. Moreover, we implemented a graphical software tool that supports the role engineering process in general and the specification of context constraints in particular. To actually enforce the context constraints that are defined on the modeling level, we extended the design and implementation of the xoRBAC component. Thereby, xoRBAC provides a flexible RBAC service that preserves the advantages of RBAC and additionally offers functions for the definition and enforcement of fine-grained context-dependent access control policies. In particular, it allows for the definition of conditional permissions.

While it is possible to define (in principle) any kind of context constraint on the modeling level, the enforcement of such constraints is clearly limited to the functionality that is provided by a concrete RBAC service. Nevertheless, the xoRBAC sensor-library can be dynamically extended with additional sensors,

and each context attribute that can be captured by an xoRBAC sensor/context function can be used in the definition of context constraints.

The xoRBAC component can be used for applications on Unix or Windows with a C or Tcl linkage [see also Neumann and Strembeck 2001]. Nevertheless, while some sensors may access platform-specific system functions we install different sensor libraries depending on the platform xoRBAC is used on. However, the sensor interfaces are (of course) platform independent. The graphical administration tool for xoRBAC is implemented with XOTcl and freely available widget sets and can be directly applied on different platforms including Unix and Windows.

The abstract design of xoRBAC is generic and can be used to extend arbitrary (traditional) RBAC services with context constraints. Our reference implementation of xoRBAC presented in this paper can be flexibly extended with reasonable efforts and thereby allows for the consideration of previously "unknown" context information. Approaches for context-dependent access control could be implemented in many other ways as the one suggested in this paper, of course (see Section 7). However, in our opinion the context constraint approach has yet a good potential to investigate the consideration and significance of context information in access control.

Novel applications using pervasive computing techniques, and the vision of ubiquitous internet access, for example, in cars or planes, yet give only a rough idea of the upcoming related security issues. Thus, we hope to increase the knowledge and understanding of *context* with respect to access control to enable the enforcement of tailored context-dependent access control policies. This is, however, a wide open ground and is likely to provide research questions for many years.

In our experiences, context constraints, as defined in this paper, are intuitively understandable and are a suitable means to model dynamic context-dependent constraints. Furthermore, they allow for the dynamic evolution of access control policies, and the alignment to changing environment conditions as they frequently occur in interactive networked environments. Although context constraints can be modeled and used in a straightforward manner, they can potentially add a great deal of complexity to access control policies. On the other hand, they add much flexibility and expressiveness, and allow for the definition of fine-grained access control policies as they are often needed in real-world applications. So far we especially gained experiences with xoRBAC in the domain of web-based collaborative applications. However, we are conducting more case studies to further improve the role engineering process and to investigate the applicability of context constraints in different application domains. For example, we are interested in the enforcement of RBAC policies that include context constraints in an ad hoc computing environment.

REFERENCES

ADAM, N., ATLURI, V., BERTINO, E., AND FERRARI, E. 2002. A content-based authorization model for digital libraries. *IEEE Trans. Knowledge Data Eng. 14*, 2 (Mar/Apr).

AHN, G. AND SANDHU, R. 2000. Role-based authorization constraints specification. *ACM Trans. Inf. Syst. Secur. 3*, 4 (Nov.).

ANTÓN, A. 1996. Goal-based requirements analysis. In *Proceedings of the IEEE International Conference on Requirements Engineering (ICRE)*.

APT, K., BLAIR, H., AND WALKER, A. 1988. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, ed. Morgan Kaufmann Publishers.

ATLURI, V. AND GAL, A. 2002. An authorization model for temporal and derived data: Securing information portals. *ACM Trans. Inf. Syst. Secur. 5*, 1 (Feb.).

BACON, J., LLOYD, M., AND MOODY, K. 2001. Translating role-based access control policy within context. In *Proceedings of the 2nd International Workshop on Policies for Distributed Systems and Networks (POLICY)*. Lecture Notes in Computer Science, 1995, Springer Verlag.

BARKLEY, J., BEZNOSOV, K., AND UPPAL, J. 1999. Supporting relationships in access control using role based access control. In *Proceedings of ACM Workshop on Role Based Access Control*.

BELOKOSZTOLSZKI, A., , EYERS, D., AND MOODY, K. 2003. Policy contexts: Controlling information flow in parameterised RBAC. In *Proceedings of the 4th International Workshop on Policies for Distributed Systems and Networks (POLICY)*.

BERTINO, E., BONATTI, P., AND FERRARI, E. 2001. TRBAC: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur. 4*, 3 (Aug.).

BERTINO, E., FERRARI, E., AND ATLURI, V. 1999. The specification and enforcement of authorization constraints in workflow management systems. *ACM Trans. Inf. Syst. Secur. 2*, 1 (Feb.).

BHARADWAJ, V. AND BARAS, J. 2003. Towards automated negotiation of access control policies. In *Proceedings of the 4th International Workshop on Policies for Distributed Systems and Networks (POLICY)*.

BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C., MALER, E., AND YERGEAU, F. 2004. Extensible Markup Language (XML) version 1.0, 3rd edition. Available at http://www.w3.org/TR/REC-xml/. W3 Consortium Recommendation.

CLARK, J. AND DEROSE, S. 1999. XML Path Language (XPath). Available at http://www.w3.org/TR/xpath. W3 Consortium Recommendation.

COHEN, E., THOMAS, R., WINSBOROUGH, W., AND SHANDS, D. 2002. Models for coalition-based access control (CBAC). In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies (SACMAT)*.

COVINGTON, M., LONG, W., SRINIVASAN, S., DEY, A., AHAMAD, M., AND ABOWD, G. 2001. Securing context-aware applications using environment roles. In *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies (SACMAT)*.

DENNING, D. 1976. A lattice model of secure information flow. *Commun. ACM 19*, 5 (May).

DEY, A. 2001. Understanding and using context. *Personal and Ubiquitous Computing 5*, 1.

EDJLALI, G., ACHARYA, A., AND CHAUDHARY, V. 1998. History-based access control for mobile code. In *Proceedings of the Fifth ACM Conference on Computer and Communications Security (CCS)*.

FERRAIOLO, D., BARKLEY, J., AND KUHN, D. 1999. A role-based access control model and reference implementation within a corporate Intranet. *ACM Trans. Inf. Syst. Secur. 2*, 1 (Feb.).

FERRAIOLO, D., SANDHU, R., GAVRILA, S., KUHN, D., AND CHANDRAMOULI, R. 2001. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur. 4*, 3 (Aug.).

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GEORGIADIS, C., MAVRIDIS, I., PANGALOS, G., AND THOMAS, R. 2001. Flexible team-based access control using contexts. In *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies (SACMAT)*.

GIURI, L. AND IGLIO, P. 1997. Role templates for content-based access control. In *Proceedings of the ACM Workshop on Role-Based Access Control*.

GUTH, S., NEUMANN, G., AND STREMBECK, M. 2003. Experiences with the enforcement of access rights extracted from ODRL-based digital contracts. In *Proceedings of the 3rd ACM Workshop on Digital Rights Management (DRM)*.

JAEGER, T. 1999. On the increasing importance of constraints. In *Proceedings of the ACM Workshop on Role-Based Access Control*.

JAEGER, T., PRAKASH, A., LIEDTKE, J., AND ISLAM, N. 1999. Flexible control of downloaded executable content. *ACM Trans. Inf. Syst. Secur. 2*, 2 (May).

JAJODIA, S., SAMARATI, P., SAPINO, M., AND SUBRAHMANIAN, V. 2001. Flexible support for multiple access control policies. *ACM Trans. Datab. Syst. 26*, 2 (June).

JARKE, M., BUI, X., AND CARROLL, J. 1998. Scenario management: An interdisciplinary approach. *Requirements Engineering Journal 3*, 3/4.

KALAM, A. E., BAIDA, R. E., BALBIANI, P., BENFERHAT, S., CUPPENS, F., DESWARTE, Y., MIÈGE, A., SAUREL, C., AND TROUESSIN, G. 2003. Organization based access control. In *Proceedings of the 4th International Workshop on Policies for Distributed Systems and Networks (POLICY)*.

KANG, M., PARK, J., AND FROSCHER, J. 2001. Access control mechanisms for inter-organizational workflow. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*.

KIM, W., GRAUPNER, S., SAHAI, A., LENKOV, D., CHUDASAMA, C., WHEDBEE, S., LUO, Y., DESAI, B., MULLINGS, H., AND WONNG, P. 2002. Web E-speak: Facilitating web-based E-services. *IEEE Multimedia 9*, 1.

LONGSTAFF, J., LOCKYER, M., CAPPER, G., AND THICK, M. 2000. A model of accountability, confidentiality and override for healthcare and other applications. In *Proceedings of the 5th ACM Workshop on Role-Based Access Control*.

MCDANIEL, P. 2003. On context in authorization policy. In *Proceedings of the ACM Symposium on Access Control Models and Technologies*.

MYLES, G., FRIDAY, A., AND DAVIES, N. 2003. Preserving privacy in environments with location-based applications. *IEEE Pervasive Comput. 2*, 1 (Jan.–Mar.).

NEUMANN, G. AND STREMBECK, M. 2001. Design and implementation of a flexible RBAC-service in an object-oriented scripting language. In *Proceedings of the 8th ACM Conference on Computer and Communications Security (CCS)*.

NEUMANN, G. AND STREMBECK, M. 2002. A Scenario-driven role engineering process for functional RBAC roles. In *Proceedings of 7th ACM Symposium on Access Control Models and Technologies (SACMAT)*.

NEUMANN, G. AND ZDUN, U. 2000. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: 7th USENIX Tcl/Tk Conference*.

NITSCHE, U., HOLBEIN, R., MORGER, O., AND TEUFEL, S. 1998. Realization of a context-dependent access control mechanism on a commercial platform. In *Proceedings of the 14th International Information Security Conference (IFIP/SEC)*.

OUSTERHOUT, J. 1994. *Tcl and the Tk Toolkit*. Addison-Wesley.

PHILLIPS, C., TING, T., AND DEMURJIAN, S. 2002. Information sharing and security in dynamic coalitions. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies (SACMAT)*.

ROLLAND, C., GROSZ, G., AND KLA, R. 1999. Experience with goal-scenario coupling in requirements engineering. In *Proceedings of the IEEE International Symposium on Requirements Engineering (RE)*.

SANDHU, R., COYNE, E., FEINSTEIN, H., AND YOUMAN, C. 1996. Role-based access control models. *IEEE Comput. 29*, 2 (Feb.).

SCHMIDT, A., BEIGL, M., AND GELLERSEN, H. 1999. There is more to context than location. *Comput. Graphics 23*, 6 (Dec.).

STREMBECK, M. 2004. Conflict checking of separation of duty constraints in RBAC—Implementation experiences. In *Proceedings of the Conference on Software Engineering (SE 2004)*.

THOMAS, R. 1997. Team-based access control (TMAC): A primitive for applying role-based access controls in collaborative environments. In *Proceedings of the ACM Workshop on Role Based Access Control*.

THOMAS, R. AND SANDHU, R. 1997. Task-based authorization controls (TBAC): A family of models for active and enterprise-oriented authorization management. In *Proceedings of the IFIP WG11.3 Conference on Database Security*.

VAN LAMSWEERDE, A. 2001. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE)*.

VAN LAMSWEERDE, A. AND LETIER, E. 2000. Handling obstacles in goal-oriented requirements engineering. *IEEE Trans. Softw. Eng. 26*, 10 (Oct.).

WANG, W. 1999. Team-and-role-based organizational context and access control for cooperative hypermedia environments. In *Proceedings of the ACM Conference on Hypertext and Hypermedia*.

WARRIOR, J., MCHENRY, E., AND MCGEE, K. 2003. They know where you are. *IEEE Spectrum 40*, 7 (July).

WEISER, M. 1991. The computer for the 21st Century. *Sci. Am. 265*, 3 (Sep.).

WEISER, M. 1993. Some computer science issues in ubiquitous computing. *Commun. ACM 36*, 7 (July).

WILIKENS, M., FERITI, S., SANNA, A., AND MASERA, M. 2002. A context-related authorization and access control method based on RBAC: A case study from the health care domain. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*.

WOLF, R. AND SCHNEIDER, M. 2003. Context-dependent access control for web-based collaboration environments with role-based approach. In *Proceedings of the 2nd International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security*. Lecture Notes in Computer Science, vol. 2776, Springer Verlag.

YAO, W., MOODY, K., AND BACON, J. 2001. A model of OASIS role-based access control and its support for active security. In *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies (SACMAT)*.

Received November 2003; revised March 2004, April 2004 and May 2004; accepted May 2004